Creation of a Contact Networks for Lions in the Serengeti National Park

Melissa Brady

May 12, 2009

Abstract

Epidemics can have a devastating effect on wildlife, particularly in populations already at risk. One such example is the spread of canine distemper virus (CDV) among lion prides of the Serengeti. An epidemic of CDV killed many lions in the Serengeti National Park in the early nineties. Network theory can be used to characterize the spread of these epidemics by adding a spatial and social structure to epidemic models. In this paper, we work to create an accurate contact network that captures the social dynamics of lion prides within the two week infectious period for CDV. For this we will use data that has been collected about the social lives of African Lions. These contact networks can then be used to run chain binomial and percolation epidemic models.

Introduction

Lion behavior plays an important role in the creation of the contact networks. As is well known, lions live together in groups called prides. A pride will consist of a group of female lions, their cubs, and a group of males. This group of males can be in charge of more then one pride; if this is the case, the group will split their time between prides (Bygott et al. 1979; Schaller 1972). As with many pack-roving animals, each pride has a territory where it spends the majority of its time. Interpride contacts often happen at the edges of these territories, due to territorial battles (Heinsohn and Packer 1995). Also important in lion population structure are nomadic lions. These lions do not have prides of their own, but rather wonder about the park; they can travel great distances throughout the park, and therefore may have an important role in spreading disease (Schaller 1972).

This contact network is set up to study the canine distemper virus (CDV). CDV is a virulent virus that attacks the central nervous system. Because it is spread by aerosol it can be transmitted when lions come into close contact, such as when fighting or sharing a carcass (Packer et al. 1999). CDV is of particular ecological and epidemiological interest. In the early nineties, an outbreak of CDV killed thirty five percent of the lions within the Serengeti, and infected eighty five percent (Roelke-Parker et al 1996). While not yet endangered, the lion is becoming more and more threatend, due to factors such as habitat loss and hunting. Therefore, it will become increasingly important to protect lion populations from disease. Because of the extensive amounts of data taken during the outbreak in the nineties, this system is a good one to model, because the results of an epidemic model can be compared against this known epidemic.

Network theory can be a very useful tool in the modeling of epidemics. Network theory was formed as a different way of looking at interacting systems; components of a system are represented by 'nodes', while interactions among components are represented as an 'edge' between the two components (Strogatz 2001). In our system, every lion will be represented by a node in the network, while an edge will represent an interaction. Because of the nature of CDV, an interaction will be defined as one lion being in close proximity with another or one lion feeding directly after another - interactions that give a good chance of transmitting the disease (Meyers et al. 2009). These networks allow us to study epidemics in populations in a mathematical manner.

Methods and Results

Creation of Contact Networks

The program to generate contact networks was coded in the scripting language Python. The goal was to get a contact network which accurately characterizes the behavior of the lions in the Serengeti National Park within a two week period. The code for the program used to generate contact networks has been attached in the supplementary material.

Pride to Pride Connections

Field data has been gathered capturing contact rates of interactions of lions in the Serengeti. This data was then fitted to give distributions, which were then used in making the contact networks. Information has been gathered concerning the latitude, longitude and pride size of each of the 25 prides modeled in this network (Craft et al. 2009). To model this system, a square grid was created that has dimensions equal to that of the study area. Each pride was placed on the appropriate location on this grid. To decide how many other prides each pride would contact, each of the 25 prides was given a list of "pride stubs" based on the a Poisson Distribution with $\lambda = 4.55$. Each stub represents a contact that pride A will have with another pride.

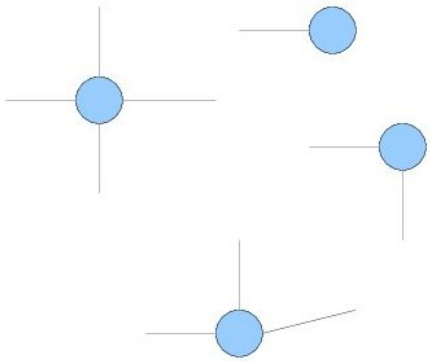

Figure 1: An example of assigning pride stubs, based on a Poisson Distribution with $\lambda = 4.55$. Here, four prides are represented by the blue nodes. Each line coming out represents an edge that will be connected to another pride edge.

To connect the prides in a realistic manner, a method for connecting pride stubs was created which uses both distance and whether the pride has recently split or not. For each pride stub, the probability of connecting to every other pride was calculated by:

$$ln(\frac{w_{c(A,B)}}{1 - w_{c(A,B)}}) = \alpha + \beta_d dt(A, B) + (-\beta_s \text{if recently split}, \beta_s \text{if not}), \quad (1)$$

where $d_t(A, B)$ is the network distance between pride A and B, and $w_{c(A,B)}$ is the probability of pride A and B having a contact. The constants are: $\alpha = 3.2652$, $\beta_d = 1.6983$, $\beta_s = 0.6961$ (Craft et al. 2009)

Because this is an agent based model, we want to network connections to be on a lion by lion basis. We use a simple method to turn a edge from pride

A to pride B to connections among the lions of pride A and the lions of B. First, a random group of lions from both prides are chosen, and these lions are fully connected. The size that the groups will be is drawn from a distribution, for which the mean is 3.65 - a distribution which was fitted from field data (Craft et al. 2009) We now have a network of prides that are interconnected via individual level connections.

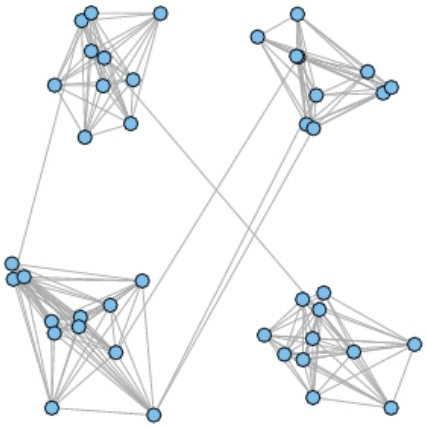

Figure 2: An example of a lion contact network. Each of the four larger clusters of nodes represents a pride. All lions of a pride are interconnected. Edges going between the prides are the interpride connections. Notice that interpride connections are perpetuated through small groups of lions from within the two prides.

Nomadic Lions

Nomadic lions, because of the large distances that they traverse, could be an important agent for spreading disease. Nomadic lions exist in groups, with their group size following a log-N distribution, with $\mu = 0.292$ and $\sigma = 0.446$.(SLP data 1985-1987) Based on radio collar data, the rather random movements of nomadic lions were found to resemble a gamma variance process (Craft 2006, unpubl. data). For our model, we ran a separate simulation to determine a distribution of how far nomadic lions travel during the two week infection period of CDV. For this simulation, a single lion was allowed to "move" in space. This was done by taking samples from the two gamma distributions, one for movement in the X direction, the other for movement in the Y direction, for a period that equated to two weeks. The distance from where the nomad started was then marked. This process was repeated 10,000 times, so that we ended up with a distribution of how far nomadic lions will travel during the infectious period of CDV.

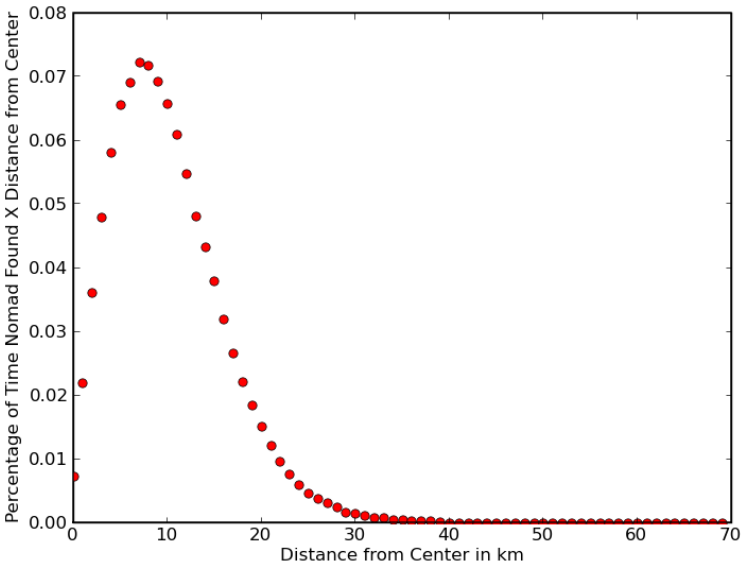

Figure 3: The distribution of nomad movement resulting from the nomad simulations.

The pride to nomad connections were made in a manner similar to pride-pride connections. The number of nomads in the park has been observed to be 180, along with 180 prides in the park (Packer 1990). The study area on which our networks are based includes only 25 prides, so 25 nomads were randomly placed onto the territory grid of our network. We had a distribution, based on field data (SLP data 1985-1987), that pride lions and nomadic lions have contacts at a frequency following a normal distribution with $\mu = 7.136$ and $\sigma = 1.018$. For each pride, a number was drawn from this distribution to create "nomad stubs": the number of nomadic groups with which that pride would have a contact. We wanted to connect lion prides to nomad groups in an intuitive manner where distance from the pride center was the main factor when considering contacts. In order to do this, for every pride to nomad stub, the distance of the pride to every nomad group was calculated. Using the results of our simulation, this distance was translated into a probability representing how likely it is that the pride will contact that particular nomad. A nomad was then randomly chosen using these probabilities. From the pride that was chosen, a random handful of lions were chosen to contact the nomad group. The method used choosing the contact group size is the same one used above for pride to pride contacts.

Results from Network Generator

Ideally, the result of our program would be a network that has distributions similar to those that have been observed in nature. This goal seems to have been met with mixed success.

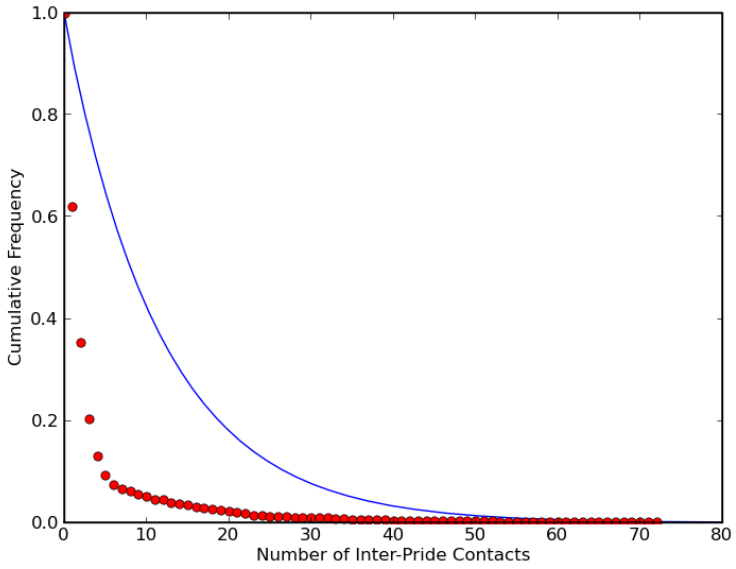

Figure 4:

The figure above shows the distribution of interpride contacts from our network generator (red circles) plotted against the function fitted for the distribution of interpride contacts observed in the field (Craft et al. 2009). As one can see, the overall shape of the distribution from our program matches that of the distribution from data. This pattern is very common in natural systems, where most individuals have few connections, while a few individuals have a large amount of connections. The distribution does not match perfectly, however, and will need future work.

Next we compare the distribution of nomad-pride connections observed in nature with those created by our network generator.

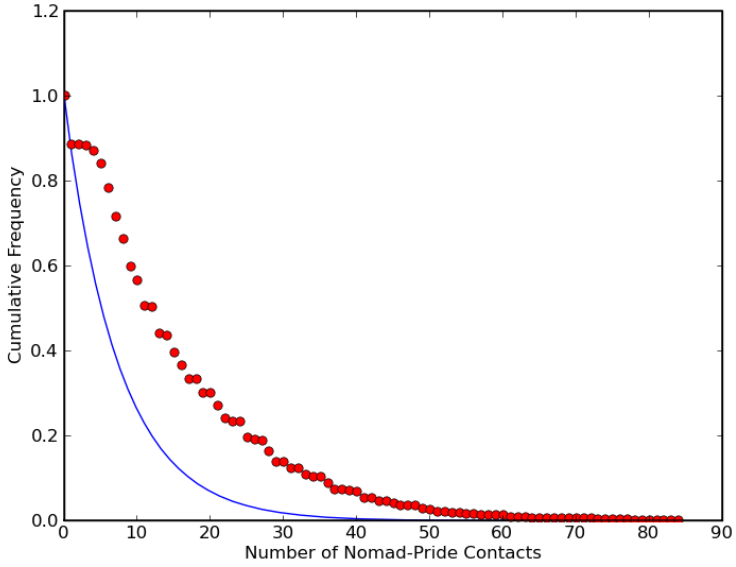

Figure 5:

Again, the overall shape of the distribution that results from the network generator matches the shape of the distribution that was fitted from field data. The distribution does not match as well as we want, so our network generator will need future work.

Discussion and Future Work

While some additional work is need to get the degree distributions of the simulation to match up with the degree simulations from field work, the networks generated from this program can be used in S-I-R disease simulations. This constitutes an important case study where an explicit contact network for disease spread can be constructed from real data. While this is important in conservation, it is also important in extending our knowledge of constructing contact networks in general. In particular, in generating human contact networks with a large number of nodes, we will need efficient algorithms for generating network from given field data.

References

Bygott, J.D., Bertram, B.C.R & Hanby, J.P. 1979 Male lions in large coalitions gain reproductive advantages. *Nature* **282**, 839-841

Craft, M., Volz, E., Packer, C., Meyers, L.A. 2009 Distinguishing epidemic waves from disease spillover in a wildlife population. *Proc.R Soc.B.*

Packer, C. 1990 Serengeti lion survey: report to TANAPA, SWRI, MWEKA and the Wildlife Division.

Packer, C., Altizer, S., Appel, M., Brown, E., Martenson, J., Lutz, H. 1999 Viruses of the Serengeti: patterns of infection and mortality in African lions. *J.Anim.Ecol.***68**, 1161-1178

Roelke-Parker, M.E. et al. 1996 A canine distemper virus epidemic in Serengeti lions (*Panthera leo*). *Nature* **379**, 441-445.

Schaller, G.B. 1972 *The Serengeti lion; a study of predator-prey relations.* Chicago, IL: University of Chicago Press.

Strogatz, S.H. 2001 Exploring complex networks. *Nature* **410**, 268-276.

Supplementary Material

Attached is the python code used to generate networks.

```
################nomadsimulation################################

import numpy.random
from math import sqrt
# Nomad Sim program: run several thousand simulations of nomad movement for a 14 day period, and use the resulting histogram as a distribution


kx=.382
ky=.714
thetax=2.85
thetay=1.743

distTrack={}
for w in range(40):
    distTrack[w]=0


denDict={}
for t in range(40):
    denDict[t]=0

def nomadDistX(x,y):
    """draws from nomad daily movement distribution"""
    xdir=numpy.random.gamma(x,y)
    return xdir

def nomadDistY(x,y):
    ydir=numpy.random.gamma(x,y)
    return ydir

def distFromZero(crd1,crd2):
    dist=sqrt(crd1**2 + crd2**2)
    return dist

def runSimOnce():
    totalUp=nomadDistX(14*kx,thetax)
    totalDown=nomadDistX(14*kx,thetax)
    totalRight=nomadDistY(14*ky,thetay)
    totalLeft=nomadDistY(14*ky,thetay)
    totalUp=totalUp
    totalDown=totalDown
    totalRight=totalRight
```

```
    totalLeft=totalLeft


    totalVert= totalUp - totalDown
    totalSide = totalRight - totalLeft

    theDist=distFromZero(totalVert,totalSide)


    #Convert this part to loop

    trac = 1
    for q in range(40):
        if(theDist < trac and theDist > trac-1):
            temp=distTrack[trac-1]
            temp2 = temp + 1
            distTrack[trac-1]=temp2

        trac=trac + 1

    return theDist

def runSimXTimes(x):
    for i in range(x):
      tempDist= runSimOnce()

def normalizeLions():
    #first add all entries in density dictionary
    hold=0
    for e in range(40):
        hold = hold + denDict[e]
    for i in range(40):
        temp=denDict[i]
        temp2=float(temp)/float(hold)
        denDict[i]=temp2



def lionsPerArea():
    """calculates the "density" of lions """
    t=0
    for b in range(40):

        denDict[b]=float(distTrack[b])/(3.14*(t+1)**2 -3.14*(t)**2)
```

```
#run the simulation
run=100000

runSimXTimes(run)
lionsPerArea()
normalizeLions()


z=0
for w in range(40):
  z=z + denDict[w]

#need to add some zeros on the end for buffering purposes:

ze=40
for z in range(30):

   denDict[ze]=0
   ze=ze+1

simDist=denDict









############################################################network generator

from networkx import *
import random as rand
from random import lognormvariate
import numpy.random
from numpy import *
import scipy as scipy
from scipy import integrate


def findMax(arr):
   theMax=0
   for i in range(len(arr)):
```

```
        temp=arr[i]
        if temp > theMax:
            theMax=temp
    return(theMax)




def assignNodeNumFemale(theVect):
    """ assigns each female lion a node number, places her in the right section of the female dictionary,
femaleDi"""
    totalFe=0
    for z in range(len(theVect)):
        num=theVect[z]
        tempVect=[]
        w=0
        while w<num:
            tempVect.append(totalFe + 1)
            totalFe=totalFe + 1
            w=w+1



        femaleDi[z]=tempVect
        #add all female nodes to lionGraph
        lionGraph.add_nodes_from(tempVect)

def assignNodeNumJuv(aVect,tot):
    """assigns each juv lion a node number; puts them in right spots in juvDi"""
    totalJu=tot
    for z in range(len(aVect)):
        num=aVect[z]
        tempVect=[]
        w=0
        while w<num:
            tempVect.append(totalJu + 1)
            totalJu=totalJu + 1
            w=w+1



        juvDi[z]=tempVect
        #add all juv nodes to lionGraph
        lionGraph.add_nodes_from(tempVect)

def fullyConnectVector(someVect):
    """fully connects a given vector: that is, give [1,2,3] returns [1,2,1,3,2,3]"""
    coi=0
```

```
    tempHolder=[]
    for i in range(len(someVect)):
        for z in range(1,len(someVect)-coi):
            somePair=(someVect[i],someVect[i+z])
            tempHolder.append(somePair)
        coi=coi+1
    return(tempHolder)

def getSharedSize():
    """returns a number from the distribution of how many prides a male coalition shares"""
    temp=rand.random()
    if temp <= .118:
        return 2
    else:
        return 1

def getCoalSize():
    """returns a number from the distribution of male coalition sizes"""
    temp1=lognormvariate(1.187,.318)
    m=int(round(temp1))
    return(m)

def getMaleNodes(num):
    """returns a list of node numbers given to the males, so that they may be correctly added to the
lionGraph and the maleDi"""
    theNodes=nodes(lionGraph)
    theNodes.sort()
    leg=len(theNodes)
    max=theNodes[leg-1]

    tempV=[]
    count=1
    for w in range(num):
        b=max + count
        tempV.append(b)
        count=count+1

    return(tempV)

def getNPride(someVect,thePride):
    """returns a vector of avalible prides that are neibhors of thePride in question"""
    return(someVect)

def removeFromVect(xVect,xNum):
    """removes a number x from a vector (not the index, but the actual number; depends on there being
no repeats"""
```

```
    xVect.remove(xNum)
    return(xVect)

def connectProb(theMales,thePrides):
    """probabalistically connects a male coalition with two prides; that is, each of the lions in the two
prides has a 1/2 chance of connecting to each male of the male coalition"""
    tempVect=[]
    for h in range(len(theMales)):
        for g in range(len(thePrides)):
            rando=rand.random()
            if rando>.5:
                male=theMales[h]
                other=thePrides[g]
                thePair=(male,other)
                tempVect.append(thePair)
    lionGraph.add_edges_from(tempVect)

def processVect(v):
    """takes the data vectors, and turns them into pride size x pride size matrix"""
    count=0
    theMatrix=[]
    for i in range(25):
        holder=v[count:(count+25)]
        theMatrix.append(holder)
        count=count + 25
    return(theMatrix)

def getPrideStubs(le):
    """returns a vector of lenght l, drawn from a poisson distriubtion with lam=9.11)"""
    while 1>0:
        stubVect=numpy.random.poisson(4.55,le)
        theSum=sum(stubVect)
        sum2=theSum%2
        if sum2==0:
            break

    return(stubVect)

def getNomadStubs(le):
    """returns a vector of lenght l, drawn from a poisson distriubtion with lam=9.11)"""
    tempVect=[0]
    for e in range(24):
        stubVect=numpy.random.normal(7.136,1.018,1)
        s=round(stubVect)
        s2=int(s)
        tempVect.append(s2)
```

```
    tempVect.remove(0)

    return(tempVect)

def distance(w1,w2):
    """calculates the network distance between pride 1 (w1) and pride 2(w2)"""
    anwr=distMatrix[w1][w2]
    return(anwr)
def recentSplit(q1,q2):
    """looks up whether or not two prides are recently split"""
    answer=splitMatrix[q1][q2]
    if answer==0:
        return -betaS
    else:
        return betaS
def calProb(p1,p2):
    """ calculates the probability of a contact between pride 1 and pride2"""
    xFun=alpha + betaD*distance(p1,p2) + recentSplit(p1,p2)
    prob=exp(xFun)/(1 + exp(xFun))
    return prob

def generateProbVect(thePride,theList):
    """generates a vector of the probability of pride a connecting to all other prides"""
    temp=[]
    for i in range(len(theList)):
        if (thePride==i) or (theList[i]==0):
            temp.append(0)
        else:
            theProb=calProb(thePride,i)
            temp.append(theProb)

    return(temp)

def pickPride(pVect):
    """chooses a pride"""
    #first, need to normalize the probablity vector
    norm=sum(pVect)
    for w in range(len(pVect)):
        pVect[w]=pVect[w]*(1/norm)

    #vector now normalized. Now need to make "stick" to toss dart on.
    probDi={}
    #fill the probablity dictionary
    tempC=0
    for g in range(len(pVect)):
        p=pVect[g]
```

```
        probDi[g]=[tempC,p+tempC]
        tempC=tempC + p
    #now, need to pick a random number, and find where it falls in the probDi dictionary. The 'key' of the
part of the dictionary on which it falls is the pride that gets choosen.
    theRandom=rand.random()
    thePride=0
    for h in range(len(probDi)): #is this the right length?
        theRange=probDi[h]
        down=theRange[0]
        up=theRange[1]
        if theRandom < up:
            thePride=h
            break

    return(thePride)

def connectPrideStubs(stubList):
    """connects the pride stubs"""

    for i in range(len(stubList)):
        while(stubList[i]>0):
            theStubs=stubList[i]
            probVect=generateProbVect(i,stubList)
            chosePride=pickPride(probVect)
# now, connect these prides, and remove both from the stubList
            thePair=(i,chosePride)
            edgeList.append(thePair)
            stubList[i]=theStubs-1
            stub2=stubList[chosePride]
            stubList[chosePride]=stub2-1
    f=sum(stubList)
    return(f)

def calculateDistance(w1,w2):
    """calculates the difference in distnace btwn the two prides"""
    dist=abs(w1-w2)
    theDist=dist/1000
    return(theDist)

def distPrideNomad(p1,n1):
    """calculates the distance btw a pride and a nomad group; p1 is the pride number, n1 is the nomad
number"""
    distP1=distanceDi[p1][0]
    distP2=distanceDi[p1][1]
    distN1=nomadDistDi[n1][0]
    distN2=nomadDistDi[n1][1]
```

```
    theDist=sqrt((distP1-distN1)**2+(distP2-distN2)**2)
    theDist=theDist/1000
    return(theDist)

def normVect(pVect):
    """normalizes a probability vector"""
    norm=sum(pVect)
    reVec=[]
    for w in range(len(pVect)):
        reVec.append(pVect[w]*(float(1)/float(norm)))

    return(reVec)

def nomadProbVect(pride):



    return(nomadVect)


def connectNomads(pride,numNomads):

    distVect=[]
    for q in range(len(nomadDi)):
        theDist=distPrideNomad(pride,q)
        theDist=round(theDist)
        distVect.append(theDist)

    #now need to convert distVect to a probability vector

    tempProb=[]
    for m in range(len(distVect)):
        temp=distVect[m]
        theValue=simDist[temp]
        tempProb.append(theValue)
    #now need to randomly pick a nomad; need to pick numNomads many nomads

    for g in range(numNomads):

        g=pickPride(tempProb)
    #now that the nomad group is picked, need to assign that nomad group to the pride
        qrs=nomadTrackerDi[pride]
        qrs.append(g)
        nomadTrackerDi[pride]=qrs
```

```
def groupSize(n):
    """returns a number from the distribution for contact group size; n is the pride size"""
    mu=.447 + .014*n

    sig=.232
    theNum=numpy.random.normal(mu,sig)

    gr=exp(theNum) -1
    gr2=round(gr)
    return(gr2)


def getPrideSize(p):
    """returns the pride size"""
    a=femaleDi[p]
    b=maleDi[p]
    c=juvDi[p]
    sizeA=len(a)
    sizeB=len(b)
    sizeC=len(c)
    theSize=sizeA + sizeB + sizeC
    return(theSize)

def selectLions(pride,size):
    """randomly selects 'size' lions from pride 'pride'"""


    bigArr=femaleDi[pride] + maleDi[pride] + juvDi[pride]
    sel=[]

    size1=int(size)
    for i in range(size1):
        sel.append(rand.choice(bigArr))

    return(sel)

def nomadSize():
    n=numpy.random.lognormal(.292,.446)
    n2=round(n)
    n3=int(n2)
    return(n3)

def inThisDiction(theDi,node):
    """looks in the dictionary array "theDi" for the node "node". It returns which pride the node was
```

```
found in. If node was not found, returns None type  """
    answer=None
    for w in range(len(theDi)):
        pickedDi=theDi[w]
        for h in range(len(pickedDi)):
            check=pickedDi[h]

            for b in range(len(check)):
                ch=check[b]

                if ch==node:
                    answer=h

    return(answer)


def diffPrides(pr1,pr2):
    """diffPrides returns a '1' if the two nodes are in different prides, a '0' if they are not (they are either
in the same pride, or one or both are nomads"""
    #sort through the maleDi/femaleDi/juvDi; when node is found, record the index of the dictionary
(this is the pride). If the node is not found, abort - the node is a nomad.

def addMales():
    """adds males to network generator"""
#now, we need to add the males

#mistake: this may break???
    avalPrides=range(len(femaleVect))

    counter = 0
    while len(avalPrides) > 0:
#for z in range(len(femaleVect)):
        numMales=getCoalSize()
        numShared=getSharedSize()
    #assign node numbers to the males
        theNodes=getMaleNodes(numMales)
        tPride=avalPrides[0]
        maleDi[tPride]=theNodes
        lionGraph.add_nodes_from(theNodes)
        avalPrides=removeFromVect(avalPrides,tPride)
        dVect=femaleDi[tPride]
        eVect=juvDi[tPride]
        fVect=dVect + eVect + theNodes
        gVect=fullyConnectVector(fVect)
        lionGraph.add_edges_from(gVect)
        if numShared==2 and len(avalPrides)>0:
```

```
        #make a list of prides that are availiable to share
            avalVect=getNPride(avalPrides,(tPride))
        #then randomly pick one of these prides
            v=int(round((len(avalVect)-1)* rand.random()))
            con=avalVect[v]
            gVect=femaleDi[con] + juvDi[con]
            maleDi[con]=theNodes
            connectProb(theNodes, gVect)
        #now remove the pride that was randomly choosen from avalPrides
            avalPrides= removeFromVect(avalPrides,con)
            counter = counter + 1

def lionConnect():



    for bw in range(len(edgeList)):
        pri1=edgeList[bw][0]
        pri2=edgeList[bw][1]
        size1=getPrideSize(pri1)
        size2=getPrideSize(pri2)
        g1=groupSize(size1)
        g2=groupSize(size2)
    #now, randomly 'take a handful' of lions to use from the prides involved
        lions1=selectLions(pri1,g1)
        lions2=selectLions(pri2,g2)
    #fully connect these selected lions
        theLions=lions1+lions2
        theConn=fullyConnectVector(theLions)
        lionGraph.add_edges_from(theConn)

def doNomads():

    for e in range(numberNomads):
    #pick a random spot for the nomad to start at
        randomLat=rand.randrange(702094,742094,1)
        randomLong=rand.randrange(9686080,9731080,1)
        nomadDistDi[e]=[randomLat,randomLong]

    nomadTrack=0
    w=0
    while nomadTrack < numberNomads:
        s=nomadSize()
        ns=getMaleNodes(s)
        nomadDi[w]=ns
        lionGraph.add_nodes_from(ns)
```

```
        nomadTrack = nomadTrack + s
        w = w + 1

    nomadVect=getNomadStubs(len(femaleVect))

    asum=sum(nomadVect)

    for e in range(len(nomadVect)):

        connectNomads(e,nomadVect[e])

    for y in range(len(nomadTrackerDi)):
        ztemp=nomadTrackerDi[y]
        ztemp.remove(0)
        nomadTrackerDi[y]=ztemp

    for k in range(len(nomadTrackerDi)):
        tempNomads=nomadTrackerDi[k]

        newNomads=[]
        for g in range(len(tempNomads)):
            kaTemp=nomadTrackerDi[k][g]

            lilArray=nomadDi[kaTemp]

            for v in range(len(lilArray)):
                finNom=lilArray[v]
                newNomads.append(finNom)

        nomadTrackerDi[k]=newNomads

    for j in range(len(nomadTrackerDi)):

        for w in range(len(nomadTrackerDi[j])):
            pSize=femaleDi[j] + maleDi[j] + juvDi[j]
            p2Size=len(pSize)
            gSize=groupSize(p2Size)
            prideLions=selectLions(j,gSize)
            nomadLions=nomadTrackerDi[j]
            theVect= prideLions + nomadLions
            nomadCon=fullyConnectVector(theVect)
            lionGraph.add_edges_from(nomadCon)

def makeNewEdge(theEdges):

#make an edge list that includes only pride lions
```

```
    newEdge=[]

    for h in range(len(theEdges)):
    #for each edge, see if both are pride lions. If so, attached to newEdge

        wq=theEdges[h][0]
        we=theEdges[h][1]

        a1=inThisDiction(dictionArray,wq)
        a2=inThisDiction(dictionArray,we)

        if a1!=None and a2!=None:
            newEdge.append(theEdges[h])
    return(newEdge)

def pridePrideEdge():

#calculate average pride to pride connections PER PRIDE: run alot

#big loop part:
    bigEdge=[]
    edgeTracker3=[]

#for each PRIDE, want to count the number of outer pride connects
    for h in range(len(femaleDi)):
        tracker=femaleDi[h]#later - set tracker = to a particular one (follow the loop)
        numEdge=0
        for q in range(len(tracker)):

            for u in range(len(newEdge)):
                newTrack=tracker[q]
                theNode1=newEdge[u][0]
                theNode2=newEdge[u][1]

        #check and see if one of the edges is "tracker"
                TF1=(theNode1==newTrack)
                TF2=(theNode2==newTrack)
                TFarr=[TF1,TF2]
                TFans=any(TFarr)

                if TFans==True:

            #now check to see if the two edges are in different prides

                    ans1=inThisDiction(dictionArray,theNode1)
                    ans2=inThisDiction(dictionArray,theNode2)
```

```
            if ans1!=None and ans2!=None and ans1!=ans2:


                numEdge=numEdge + 1

        edgeTracker3.append(numEdge)

    return(edgeTracker3)


def nodeNodeEdge():
#calculating average outer pride connections PER NODE
    edgeTracker=[]
    #intitalize edgeTracker with 0's
    for u in range(len(allPrideNodes)):
        edgeTracker.append(0)

    print "reached function"
    for h in range(len(allPrideNodes)):
        tracker=allPrideNodes[h]#set tracker = to a particular one (follow the loop)

        for u in range(len(newEdge)):
            theNode1=newEdge[u][0]
            theNode2=newEdge[u][1]


        #check and see if one of the edges is "tracker"
            TF1=(theNode1==tracker)
            TF2=(theNode2==tracker)
            TFarr=[TF1,TF2]
            TFans=any(TFarr)



            if TFans==True:

            #now check to see if the two edges are in different prides

                ans1=inThisDiction(dictionArray,theNode1)
                ans2=inThisDiction(dictionArray,theNode2)

                if ans1!=None and ans2!=None and ans1!=ans2:
                    temp = edgeTracker[h] + 1
                    edgeTracker[h]=temp
```

```
    return(edgeTracker)

def nomadPrideEdge():

#calculating avegage pride-nomad connections:



    edgeTracker4=[]
#for each PRIDE, want to count the number of outer pride connects
    for h in range(len(femaleDi)):
        tracker=femaleDi[h]
        numEdge=0
        for q in range(len(tracker)):

            for u in range(len(theEdges)):
                newTrack=tracker[q]
                theNode1=theEdges[u][0]
                theNode2=theEdges[u][1]


        #check and see if one of the edges is "tracker"
                TF1=(theNode1==newTrack)
                TF2=(theNode2==newTrack)
                TFarr=[TF1,TF2]
                TFans=any(TFarr)

                if TFans==True:

            #now check to see if one of the edges is a nomad.

                    ans1=inThisDiction([nomadDi],theNode1)
                    ans2=inThisDiction([nomadDi],theNode2)
                    #also need to check if nomads are in some group



                    if (ans1!=None or ans2!=None) and ans1!=ans2:
                        numEdge=numEdge + 1


        edgeTracker4.append(numEdge)

    return(edgeTracker4)
```

```
#Constants:
alpha=3.2652
betaD=1.6983
betaS=.6961
nbVect=[0,2,2,2,2,1,2,1,1,2,1,2,1,2,2,2,2,1,2,1,2,2,2,2,2,2,0,1,1,2,2,2,1,2,2,2,2,2,2,1,2,2,2,1,2,1,2,2,1,2,2,1,
0,1,2,2,2,1,2,2,2,2,2,2,1,2,2,2,1,2,1,2,2,1,2,2,1,1,0,2,2,2,1,2,2,2,2,2,2,1,2,2,2,1,2,1,2,2,1,2,2,2,2,2,0,1,1,2,2,
2,2,2,2,2,2,2,2,1,2,2,2,2,2,2,2,1,2,2,2,1,0,1,1,2,2,2,2,1,1,2,2,1,1,2,1,2,1,2,2,1,2,2,2,2,1,1,0,2,2,2,2,2,2,2,2,2,
2,1,2,1,2,2,2,2,1,1,1,1,1,2,1,2,0,1,1,1,1,1,2,2,2,2,1,1,1,1,2,2,1,2,1,2,2,2,2,2,2,1,0,2,1,2,1,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,1,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,1,2,1,2,2,2,2,2,2,1,1,2,0,1,1,2,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,1,2,2,1,0,2,2,2,2,2,2,1,2,1,2,2,2,2,1,2,2,2,2,1,2,1,1,2,1,2,0,2,2,1,2,1,2,1,2,2,2,2,1,2,2,2,2,2,1,2,2,2,2,2,2,
2,0,2,2,2,2,2,2,2,2,1,2,2,2,1,1,1,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,1,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,1,2,2,0,1,1,
2,2,2,2,1,2,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,2,1,0,1,2,2,2,1,1,2,1,1,2,2,2,1,1,1,1,2,2,1,2,1,2,2,1,1,0,2,1,2,1,2,2,1,2
,1,1,1,2,2,2,1,2,2,2,1,2,2,2,2,2,2,0,2,1,2,2,2,2,1,2,2,2,2,1,1,1,2,2,2,2,1,2,2,2,2,1,2,0,2,2,2,2,1,2,1,1,1,2,2,2,1,
2,2,2,1,2,2,1,2,2,2,1,2,0,2,2,1,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,2,1,1,2,2,2,0,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,
2,1,1,2,2,2,2,1,0,2,2,2,1,1,1,2,2,2,1,2,1,2,2,2,2,1,2,2,2,2,2,1,2,2,0,2,2,2,2,2,2,1,1,2,2,2,2,2,1,2,2,2,1,1,2,1,2,
2,2,2,0]

reSplit=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

splitMatrix=processVect(reSplit)
distMatrix=processVect(nbVect)

femaleVect=[1,6,3,2,3,2,1,4,4,11,7,5,3,1,5,5,2,1,1,10,3,2,1,5,0]
juvVect=[0,13,1,2,1,0,1,1,0,10,4,1,0,0,0,0,1,1,2,2,3,2,0,9,0]
totalFem=sum(femaleVect)

femaleDi={}
maleDi={}
juvDi={}

numberNomads=25
kx=.382
```

```
thetax=2.85
ky=.714
thetay=1.743

distanceDi={}
#this is a dictionary of where all the prides are on the map

distanceDi[0]=[742094,9710080]
distanceDi[1]=[713094,9719080]
distanceDi[2]=[701094,9724080]
distanceDi[3]=[705094,9730080]
distanceDi[4]=[702094,9732080]
distanceDi[5]=[733094,9696080]
distanceDi[6]=[736094,9690080]
distanceDi[7]=[728094,9710080]
distanceDi[8]=[715094,9721080]
distanceDi[9]=[715094,9723080]
distanceDi[10]=[714094,9731080]
distanceDi[11]=[708094,9721080]
distanceDi[12]=[703094,9725080]
distanceDi[13]=[713094,9718080]
distanceDi[14]=[722094,9686080]
distanceDi[15]=[705094,9735080]
distanceDi[16]=[703094,9710080]
distanceDi[17]=[707094,9697080]
distanceDi[18]=[709094,9702080]
distanceDi[19]=[718094,9704080]
distanceDi[20]=[700094,9728080]
distanceDi[21]=[723094,9715080]
distanceDi[22]=[704094,9730080]
distanceDi[23]=[713094,9704080]
distanceDi[24]=[708094,9732080]
distanceDi[25]=[733094,9710080]


########really where network generator starts########

#for histogram of pride-pride connections per pride
reallyBigEdge=[]
#for histogram of pride-pride connections per node
reallyBigEdge2=[]
#nomad-pride distribution
reallyBigEdge3=[]


howManyRuns=50
```

```
for u in range(howManyRuns):

#network of all connection
   lionGraph=Graph()
   nomadDi={}
   nomadDistDi={}
   nomadTrackerDi={}

#initialize nomadTrackerDi
#assign females a node number
   assignNodeNumFemale(femaleVect)
   assignNodeNumJuv(juvVect,totalFem)

#get corresponding females from femaleDi and juvs from juvDi into one vector
   for e in range(len(femaleDi)):
      fVect=femaleDi[e]
      jVect=juvDi[e]
      bVect=fVect + jVect
      cVect=fullyConnectVector(bVect)
#now add edges to the graph
      lionGraph.add_edges_from(cVect)
#now juvenilles and females of one pride are fully connected

   for u2 in range(len(femaleVect)):
      nomadTrackerDi[u2]=[0]

   addMales()

   while 2>1:
      edgeList=[]
      prideStubs=getPrideStubs(len(femaleVect))
      qu=connectPrideStubs(prideStubs)
      if qu ==0:
         break

   lionConnect()
   doNomads()

#network statistics:

   theEdges=lionGraph.edges()
   dictionArray=[maleDi,femaleDi,juvDi]
#need to make a big node list of all the nodes to check

   allMale=[]
   for t in range(len(maleDi)):
```

```
        tempMaleDi=maleDi[t]
        allMale=allMale + tempMaleDi
    allFemale=[]

    for t in range(len(femaleDi)):
        tempFemaleDi=femaleDi[t]
        allFemale=allFemale + tempFemaleDi
    allJuv=[]

    for t in range(len(juvDi)):
        tempJuvDi = juvDi[t]
        allJuv=allJuv + tempJuvDi
    allPrideNodes=allMale + allFemale + allJuv

    newEdge=makeNewEdge(theEdges)

    edgeTracker3=pridePrideEdge()
    reallyBigEdge=reallyBigEdge + edgeTracker3

    edgeTracker2=nomadPrideEdge()
    reallyBigEdge3=reallyBigEdge3 + edgeTracker2

    edgeTrackerNN=nodeNodeEdge()
    reallyBigEdge2 = reallyBigEdge2 + edgeTrackerNN

#######DONE with network generator!!!#############


#calculate percents for reallyBigEdge


max1=findMax(reallyBigEdge)

pCount=[]
#initialize pCount
for h in range(max1+1):
    pCount.append(0)

for b in range(len(reallyBigEdge)):

    for w in range(max1+1):

        if w == reallyBigEdge[b]:

            someT=pCount[w]
            someT=someT + 1
```

```
            pCount[w]=someT

#now needs to make pCount into a prob vect:
theL = len(pCount)
for g in range(theL):
    wTemp=pCount[g]
    qTemp=float(wTemp)/float(25*howManyRuns)
    pCount[g]=qTemp



#calculate percents for reallyBigEdge3 (pride nomad)


max2=findMax(reallyBigEdge3)

pCount2=[]
#initialize pCount
for h in range(max2+1):
    pCount2.append(0)

for b in range(len(reallyBigEdge3)):

    for w in range(max2+1):

        if w == reallyBigEdge3[b]:

            someT=pCount2[w]
            someT=someT + 1
            pCount2[w]=someT

#now needs to make pCount into a prob vect:
theL2 = len(pCount2)
for g in range(theL2):
    wTemp=pCount2[g]
    qTemp=float(wTemp)/float(25*howManyRuns)
    pCount2[g]=qTemp



#calculate percents for reallyBigEdge2 (pride pride per node)


max3=findMax(reallyBigEdge2)

pCount3=[]
```

```
#initialize pCount
for h in range(max3+1):
   pCount3.append(0)

for b in range(len(reallyBigEdge2)):

   for w in range(max3+1):

      if w == reallyBigEdge2[b]:

         someT=pCount3[w]
         someT=someT + 1
         pCount3[w]=someT

#now needs to make pCount into a prob vect:
theL3 = len(pCount3)
for g in range(theL3):
   wTemp=pCount3[g]
   qTemp=float(wTemp)/float(25*howManyRuns)
   pCount3[g]=qTemp
```